

# **THIS PAGE IS INSERTED BY OIPE SCANNING**

**IMAGES WITHIN THIS DOCUMENT ARE BEST AVAILABLE COPY AND CONTAIN DEFECTIVE IMAGES SCANNED FROM ORIGINALS SUBMITTED BY THE APPLICANT.**

**DEFECTIVE IMAGES COULD INCLUDE BUT ARE NOT LIMITED TO:**

**BLACK BORDERS**

**TEXT CUT OFF AT TOP, BOTTOM OR SIDES**

**FADED TEXT**

**ILLEGIBLE TEXT**

**SKEWED/SLANTED IMAGES**

**COLORLED PHOTOS**

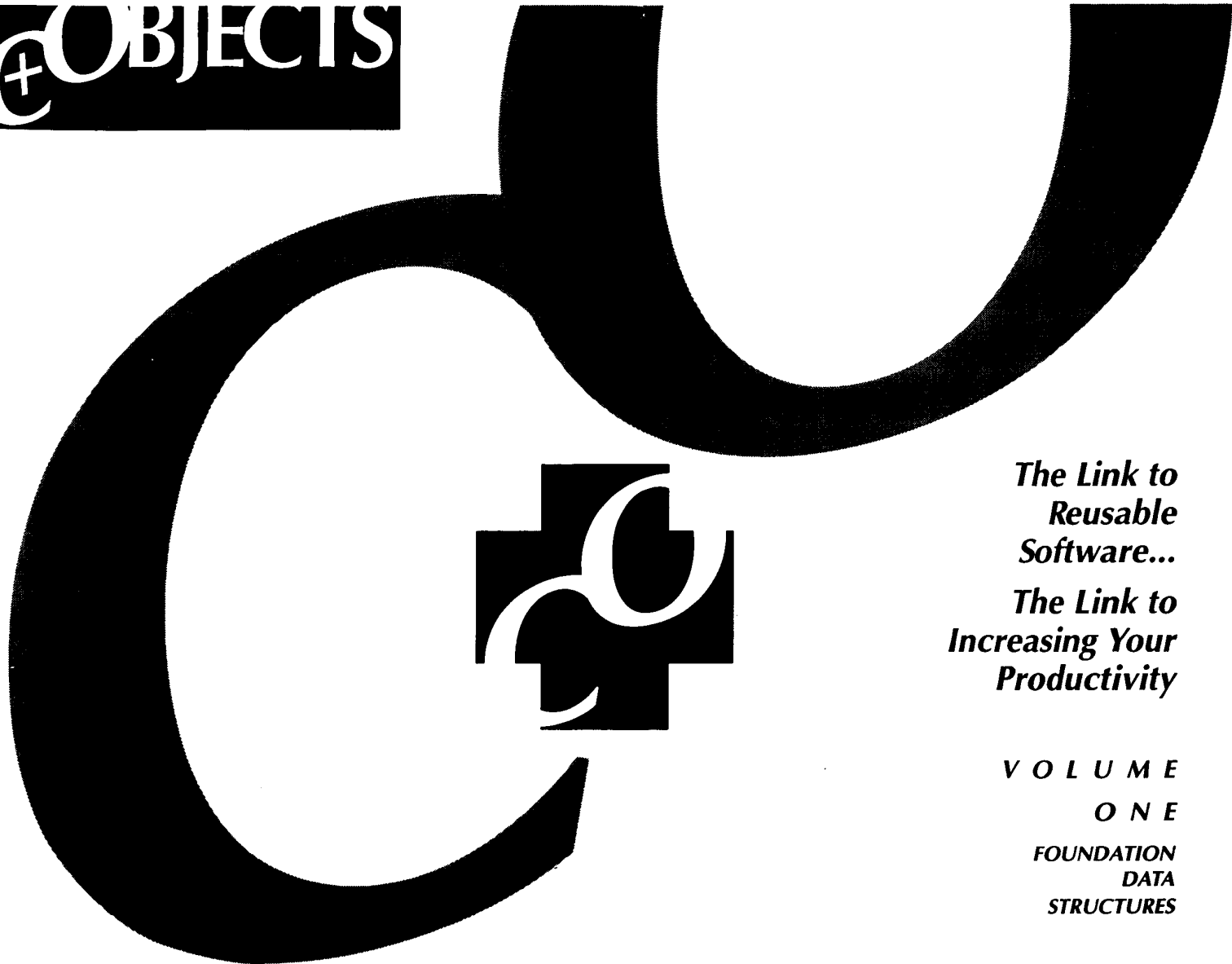
**BLACK OR VERY BLACK AND WHITE DARK PHOTOS**



**GRAY SCALE DOCUMENTS**

**IMAGES ARE BEST AVAILABLE COPY.  
RESCANNING DOCUMENTS *WILL NOT*  
CORRECT IMAGES.**



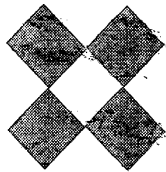
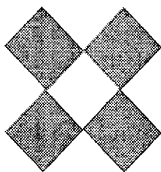


## VOLUME

## ONE

# FOUNDATION DATA STRUCTURES

[illegible]



## VOLUME 1 PRODUCT FEATURES

- *Trees, Linked-Lists, Dynamic Arrays, Graphs, Strings, Dates, Objects, Classes*
- *Object-oriented design and implementation*
- *Written entirely in C*
- *Derive your own object types: Symbol Tables, Graphical Object Lists, Pars Trees, etc.*
- *Professional, fully tested code*
- *Advanced, multi-level exception-handler speeds coding and debugging*
- *Educational tool for data structures, object-oriented programming techniques and software engineering*

## OOP FEATURES

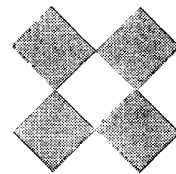
- *Each "class" is a C structure with related functions*
- *Objects are fully encapsulated*
- *Static and dynamic binding of "messages"*
- *New classes can "inherit" functionality and data from multiple object types*
- *Object-oriented compiler source code*

## WHAT'S INCLUDED

- 14 types of object, over 300 functions
- User's Guide explains object-oriented programming techniques, deriving your own object types, and includes tutorials
- Reference Guide with detailed information on each object type and function
- Demo and example programs
- Full source code available as option
- Debugging and production versions of libraries
- Support hot-line
- 30 day, money back guarantee



***C+OBJECTS™ is a portable,  
object-oriented C function  
library used to reduce  
the investment required  
to build complex software.***



**What can C+OBJECTS  
do for me?**

It can give you more creative time to design programs because you'll spend less time coding and debugging them. That's because the fundamental data structures used in many programs have already been built for you. Volume 1 includes data structures such as **doubly-linked lists, trees, dynamic arrays and graphs**. Volume 2 includes additional data structures such as **outlines, hash tables, stacks and queues** (details on Volume 2 appear in a separate brochure).

Your programs will be more reliable with the sophisticated, multi-level exception-handler and debug libraries. You also get Julian (date) and String object

oriented format. The Julian routines have many calculations not available in other products.

**Can C+OBJECTS data  
structures be custom-  
ized?**

That's the whole idea! Customizing and extending the functions of C+OBJECTS data structures is simple. Just "inherit" functionality from one or more C+OBJECTS data structures and add your own code and data on top.

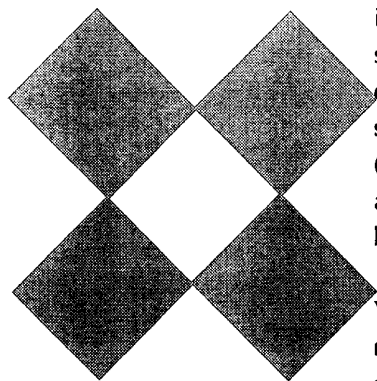
For example, you could use the Tree data structure as the foundation for a parse tree. Or you might build a data structure for maintaining a graphical display list using the Doubly-Linked List object type. If you were building a data-flow diagram editor as part of a CASE package,

you would find Graph, Vertex, and Edge well suited to the task. The uses for C+OBJECTS structures are virtually unlimited!

Customizing or extending C+OBJECTS object types does not involve modifying or recompiling the C+OBJECTS code or structures. C+OBJECTS would not be a useful tool otherwise.

**What do you mean by  
an "object-oriented"  
function library?**

Just as structured programming and structured design principles are not language dependent, neither are the principles of object-oriented programming. When we designed C+OBJECTS, we took the fundamental object-oriented programming techniques and applied them to C. Other object-oriented



tools for C have mimicked the Smalltalk implementation, complete with all of Smalltalk's faults and inefficiencies—we didn't, we married the best of both worlds.

#### And Performance?

C+OBJECTS is written entirely in C and does not use pre-processors or interpreters. Performance is what sets C+OBJECTS apart from the others.

C+OBJECTS provides macros for many functions. This gives you all the advantages of encapsulation without the performance penalty of calling a function to do a simple task.

Additionally, the messaging and inheritance features are implemented in a manner *tailor-made for C*. The result is cleaner and more efficient than Smalltalk's mechanisms.

#### Can it help me debug my programs faster?

Yes! C+OBJECTS advanced debugging features allow you to create *reliable* programs and do so easier and more quickly than you thought possible.

First, C+OBJECTS uses function prototypes to catch simple errors at compile time involving incorrect type, wrong ordering or wrong number of parameters.

Second, C+OBJECTS can detect when it is being passed a NULL or uninitialized pointer, pointers to the wrong type, or pointers to structures which have been "garbaged". It also checks for illegal values in other parameters types.

Third, C+OBJECTS includes an advanced exception handler package. With it, you can set up a single (or multiple level) exception handler which traps exceptions generated by C+OBJECTS functions.

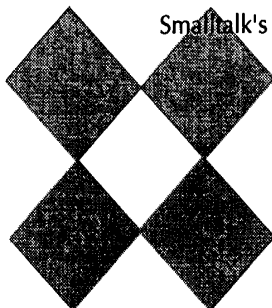
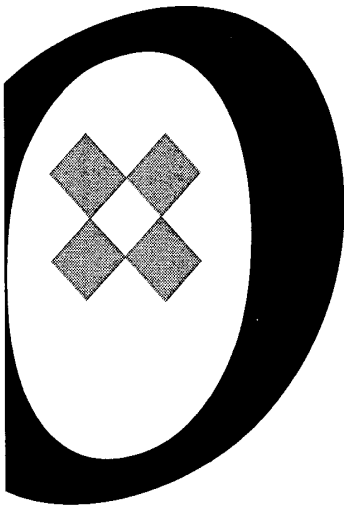
If an exception is raised, you can determine the type and where it occurred. You can then recover from the exception or abort, depending on which is most appropriate. Exception handlers can allow your program to be well behaved, even in the presence of bugs.

*This advanced error detection technique can be used in your own code as well.* No longer do your programs need to check status codes after each function call. This results in less coding yet more reliable programs.

Once your program has been debugged, you can use C+OBJECTS Production Libraries with macro functions. This eliminates most or all of the debugging checkpoints.

#### What else can it do to increase my productivity?

C+OBJECTS goes beyond conventional function libraries by supplying a complete set



---

of *object-oriented control-structures*.

These functions allow you to traverse data structures *without* having to use for, while, or do-while statements.

Control-structure functions simplify programs and eliminate a large number of potential errors — boundary conditions in loops for example.

Control-structure functions call a function of your choice for each item traversed. You can “inherit” these control-structure functions in your own data structures or create your own.

#### **How portable is C+OBJECTS?**

C+OBJECTS was designed for portability to any operating system. Expect to see versions for Windows, OS/2, Presentation Manager, and Macintosh soon.

#### **Is it suitable as an educational tool?**

Yes. As an educational aid, it can teach you the principles of object-oriented programming. The User's Guide explains object-oriented programming and the differences between C+OBJECTS and Smalltalk. It could even be used as a primer for C programmers who wish to understand more about Smalltalk.

It can teach students the concepts of abstract data types and basic data structures. The linked list, tree, and graph types could form the foundation for a data structures class.

A software engineering course would benefit from a study of C+OBJECTS. It demonstrates good design principles, strict naming and portability conventions, and *defensive programming* techniques.

But don't let this fool you into thinking C+OBJECTS is *only* of

educational value.

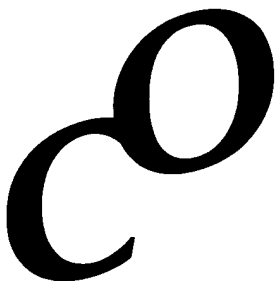
C+OBJECTS is a serious development tool for professionals.

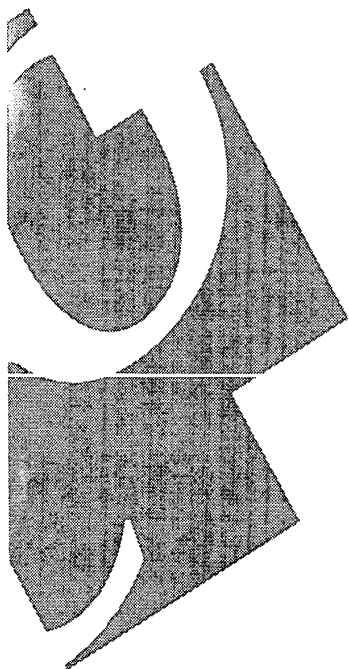
#### **What about source code, royalties etc.?**

Full source code is available as an option. You will get more educational value out of C+OBJECTS with the source, but you don't need it to fully use or understand the product. Source will of course be necessary if you are porting C+OBJECTS to a new environment — call us first though, we may be able to help.

There are no royalties on programs developed using C+OBJECTS Volumes 1 or 2 and we do not require you to reproduce our copyright notice on your programs.

Call us for information on volume pricing, site licensing, and educational discounts.



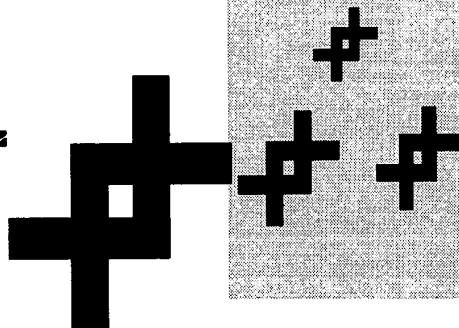


## Class

A **Class (Cls)** implements the object-oriented properties inheritance and messaging. It is used to subclass another object type. (See also **Object** page 11)

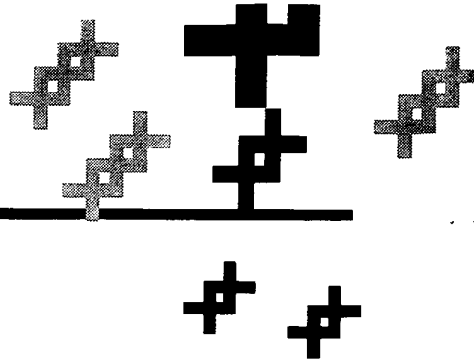
## Doubly Linked List

A **Doubly-Linked List (Dll)** object is used to represent the head and tail of a linked list. A **Dll** contains objects of type **List Element** (or derivative objects). (See also **List Element** page 10)



<b>ClsDefaultInit</b>	Initialize using defaults
<b>ClsDestroyObj</b>	Deallocate object
<b>ClsGetClientOffset</b>	Return client offset
<b>ClsGetGpMsgFunc</b>	Return (ptr.) message function ptr.
<b>ClsGetMsgFunc</b>	Return (int) message function ptr.
<b>ClsInit</b>	Initialize the class
<b>ClsNewObj</b>	Allocate object
<b>ClsSetClientOffset</b>	Set client offset
<b>ClsSendObjMsg</b>	Send message, return int
<b>ClsSendObjGpMsg</b>	Send message, return pointer

<b>DllAppend</b>	Append element(s) to list
<b>DllAppendLast</b>	Append element(s) to end of list
<b>DllAppendOne</b>	Append one element to list
<b>DllAsObj</b>	Return list as object
<b>DllClear</b>	Clear list
<b>DllClient</b>	Return client of list
<b>DllClientDo</b>	Do function: all elements
<b>DllClientDoBkws</b>	Do function: elements backwards
<b>DllClientCount</b>	Do function: count elements
<b>DllClientFind</b>	Do search function: all elements
<b>DllClientFirst</b>	Return client of first element
<b>DllClientGetNth</b>	Return Nth client
<b>DllClientOrNull</b>	Return client or null
<b>DllClientLast</b>	Return client of last element
<b>DllCut</b>	Cut element(s) from list
<b>DllCutAll</b>	Cut all elements from list
<b>DllCutOne</b>	Cut one element from list
<b>DllDeInit</b>	Deinitialize list
<b>DllDestroy</b>	Deinitialize list, free space
<b>DllGetFirst</b>	Return first element
<b>DllGetLast</b>	Return last element
<b>DllGetNth</b>	Return Nth element
<b>DllIsEmpty</b>	Is list empty?
<b>DllInit</b>	Initialize list
<b>DllInsert</b>	Insert element(s) in list




---

## Doubly Linked List

## Dynamic Pointer Array

*Dynamic Pointer Arrays (Dpa) are useful for storing arrays of pointers to objects of any type. A Dpa is dynamic because storage for the array is allocated and reallocated dynamically as the size of the array changes*

<i>DllInsertFirst</i>	Insert elements first
<i>DllInsertOne</i>	Insert element in list
<i>DllMakeFirst</i>	Make element first
<i>DllMakeLast</i>	Make element last
<i>DllNew</i>	Initialize list object, allocate space
<i>DpaAppend</i>	Append an element
<i>DpaCut</i>	Delete element(s)
<i>DpaClear</i>	Clear dynamic array
<i>DpaCountTrueReturns</i>	Do function: count True returns
<i>DpaDeInit</i>	Deinitialize dynamic array
<i>DpaDestroy</i>	Deinitialize object, free all memory
<i>DpaDo</i>	Do function: all elements
<i>DpaDoRange</i>	Do function: range of elements
<i>DpaDoRangeCheckRet</i>	Do function: range, check return
<i>DpaDoRegion</i>	Do function: region of elements
<i>DpaDoSelfAndSuccessors</i>	Do function: successors
<i>DpaFindBkwd</i>	Find index returning True
<i>DpaFindFrwd</i>	Find index returning True
<i>DpaFindPtrBkwd</i>	Find index with matching pointer
<i>DpaFindPtrFrwd</i>	Find index with matching pointer
<i>DpaFindRangeFrwd</i>	Find index returning True for range
<i>DpaFindRangeBkwd</i>	Find index returning True for range
<i>DpaGetLast</i>	Return last element in array
<i>DpaGetNth</i>	Return Nth array element
<i>DpaGetSize</i>	Return number of elements
<i>DpaInit</i>	Initialize dynamic array object
<i>DpaLoad</i>	Load array by looping function
<i>DpaMakeElementsZero</i>	Make range of elements null
<i>DpaNew</i>	Initialize object and allocate space
<i>DpaPaste</i>	Paste element(s) into array
<i>DpaScrollDown</i>	Scroll down N lines in array
<i>DpaScrollUp</i>	Scroll up N lines in array
<i>DpaSetNth</i>	Set Nth element of array
<i>DpaSetSize</i>	Set array size to N elements



## Edge

An *Edge (Edg)* is used to represent a directed edge in a *Graph (Grf)*. An edge can be connected and disconnected from two vertices (*Vtx*). An edge can belong to a single graph. (See also *Vertex* page 14 and *Graph* page 9)

## Exception

An *Exception (Exc)* is a container for error/status information used when a program wants to raise an exception. An *Exc* contains the type of error, its location, and other pertinent information. Exceptions are invoked via a *Thread (Thr)*. (See also *Threads* page 12)

<i>EdgClientDo</i>	Do function: edge
<i>EdgConnectToVertices</i>	Connect edge to vertices
<i>EdgConnectToGrf</i>	Connect edge to graph
<i>EdgCompareInVtx</i>	Compare vertex to incoming edge
<i>EdgCompareOutVtx</i>	Compare vertex with outgoing edge
<i>EdgDeInit</i>	Deinitialize the edge object
<i>EdgDisconnectFromGrf</i>	Disconnect edge from graph
<i>EdgGetClient</i>	Return client of edge
<i>EdgGetGraphLel</i>	Return as graph list element
<i>EdgGetGrf</i>	Return graph
<i>EdgGetInLel</i>	Return incoming edge list element
<i>EdgGetInVtx</i>	Return incoming vertex
<i>EdgGetNextIn</i>	Return next incoming edge
<i>EdgGetNextOut</i>	Return next outgoing edge
<i>EdgGetOutLel</i>	Return outgoing edge list element
<i>EdgGetOutVtx</i>	Return outgoing vertex
<i>EdgGetVertices</i>	Return vertices to edge
<i>EdgInit</i>	Initialize the edge object
<i>EdgInGrf</i>	Is edge in graph?
<i>EdgNew</i>	Initialize edge object and allocate
<i>EdgSendDestroy</i>	Send message for vertex destruction
<i>EdgUpdateInVtx</i>	Replace incoming vertex
<i>EdgUpdateOutVtx</i>	Replace outgoing vertex

<i>ExcClear</i>	Clear exception
<i>ExcDeInit</i>	Deinitialize exception
<i>ExcDestroy</i>	Deinitialize exception, free space
<i>ExcGetCode</i>	Return error code
<i>ExcGetFile</i>	Return file where error detected
<i>ExcGetLine</i>	Return line where error detected
<i>ExcGetOpSysErr</i>	Return system error code
<i>ExcGetType</i>	Return type of error
<i>Exclnit</i>	Initialize exception
<i>ExcIsFatal</i>	Is exception non-recoverable?
<i>ExcNew</i>	Initialize exception, allocate space
<i>ExcSet</i>	Set exception fields

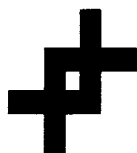
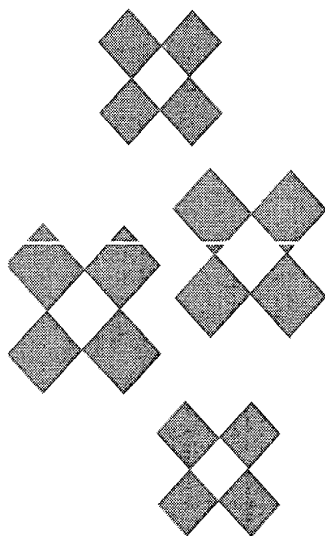
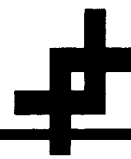
## Graph

A **Graph (Grf)** object is used to represent a directed graph (or digraph) as understood by graph theory. A Graph is a collection of Vertices (Vtx) and (directed) Edges (Edg). A graph can be sorted topologically to determine if it is acyclic. (See also Vertex page 14 and Edge page 8)

## Julian Date

A **Julian Date (Jul)** is used to represent a specific day in a specific year. The representation is purposely made explicit by its name. This representation of dates is most appropriate when date calculations are of more interest than formatting.

<b>GrfAddEdg</b>	Add edge to graph
<b>GrfAddVtx</b>	Add vertex to graph
<b>GrfClear</b>	Clear the graph
<b>GrfCountEdg</b>	Count edges of graph
<b>GrfCountVtx</b>	Count vertices of graph
<b>GrfDestroy</b>	Deinitialize graph and free space
<b>GrfDeInit</b>	Deinitialize graph object
<b>GrfDoEdgClient</b>	Do function: edges
<b>GrfDoVtxClient</b>	Do function: vertices
<b>GrfDoVtxInTopOrder</b>	Do function: vertices in topological order
<b>GrfInit</b>	Initialize graph object
<b>GrfNew</b>	Initialize graph allocate space
<b>GrfRemoveEdg</b>	Remove edge from graph
<b>GrfRemoveVtx</b>	Remove vertex from graph
<b>GrfTopologicalSort</b>	Do topological sort of graph
<b>JulAddDays</b>	Add/subtract days to date
<b>JulAddDaysL</b>	Add/subtract days to date (long)
<b>JulAddMonths</b>	Add/subtract months to date
<b>JulAddQuarters</b>	Add/subtract quarters to date
<b>JulAddYears</b>	Add/subtract years to date
<b>JulToCalendar</b>	Julian day to day, month, year
<b>JulCopy</b>	Copy julian day
<b>JulDateStrToJulian</b>	Date string to julian day
<b>JulDaysInMonth</b>	Days in month
<b>JulDaysInQuarter</b>	Days in quarter
<b>JulDaysInYear</b>	Days in year
<b>JulDayOfYear</b>	Day number in year
<b>JulDayOfWeek</b>	Day number in week
<b>JulDiff</b>	Days between two dates
<b>JulDiffL</b>	Days between two dates (long)
<b>JulFromCalendar</b>	Day, month, year to julian
<b>JulGetMaxValue</b>	dayReturn maximum julian value
<b>JulGetSystemJulianDay</b>	System date as julian day
<b>JulInit</b>	Initialize object
<b>JulIsLeapYear</b>	Is date in leap year?
<b>JulIsMaxValue</b>	Is date maximum julian value?



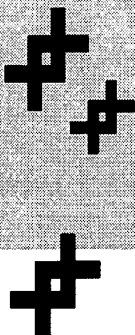
## Julian Date

A Julian Date (Jul) is used to represent a specific day in a specific year. The representation is purposely made explicit by its name. This representation of dates is most appropriate when date calculations are of more interest than formatting. (Continued from page 9)

## List Element

A List Element (Lel) object is used to maintain membership in a doubly-linked list (Dll). A Lel knows its previous and next list elements and the list it belongs to, if any. (See also Doubly-Linked List page 6)

<b>JulMax</b>	The maximum of two julian dates
<b>JulMin</b>	The minimum of two julian dates
<b>JulMonthDayDiff</b>	Days between date and day/month
<b>JulMonthString</b>	Fill string with month and year
<b>JulQuarterString</b>	Fill string with quarter and year
<b>JulSameDayMonth</b>	Are dates same day and month?
<b>JulSetMaxDate</b>	Set date to maximum value
<b>JulToDateStr</b>	Fill date string with specified format
<b>JulValidateDate</b>	Validate date passed as string
<b>JulWeekString</b>	Fill string with week
<b>JulYearString</b>	Fill string with year
<b>LelAppend</b>	Append element(s) to list
<b>LelAsObj</b>	Return element as object
<b>LelClientDll</b>	Return client of list
<b>LelClientNext</b>	Return client of next element
<b>LelClientPrev</b>	Return client of previous element
<b>LelClientCountSelfAndSuccessors</b>	Return count of successors
<b>LelClientDoSelfAndPredecessors</b>	Do function: predecessors
<b>LelClientDoSelfAndSuccessors</b>	Do function: successors
<b>LelClientDoPredecessors</b>	Do function: predecessors
<b>LelClientDoSuccessors</b>	Do function: successors
<b>LelClientDoRange</b>	Do function: range
<b>LelClientFindRange</b>	Do search function: range
<b>LelCount</b>	Count elements
<b>LelCut</b>	Cut element(s) from list
<b>LelDeInit</b>	Deinitialize list element object
<b>LelDoRange</b>	Do function: for range
<b>LelElementsAreInOrder</b>	Are two elements in order?
<b>LelGetClient</b>	Return client
<b>LelGetDll</b>	Return list object is in
<b>LelGetNthSuccessor</b>	Return Nth successor element
<b>LelGetNext</b>	Return next element
<b>LelGetPrev</b>	Return previous element
<b>LelInit</b>	Initialize list element object
<b>LelInList</b>	Is element in list?
<b>LelInsert</b>	Insert element(s) to list
<b>LelMakeList</b>	Make elements into list



## Object

An *Object (Obj)* implements the object-oriented properties of inheritance and messaging. It is of use for implementing reusable data types (as opposed to application-specific types). (See also *Class* page 6)

## String

The *String (Str)* class is used to represent null terminated character arrays.

## Task

A *Task (Tsk)* object is used to represent a program. A *Tsk* owns all the threads in that task (one in MS-DOS). A task contains information used to invoke the program and other global information which belongs to a task. (See also *Thread* page 12 and *Exception* page 8)

<i>ObjDeInit</i>	Deinitialize object
<i>ObjDestroyClient</i>	Deallocate object
<i>ObjGetClientOrNull</i>	Return client
<i>ObjGetGpMsgFunc</i>	Return (ptr.) message function
<i>ObjGetMsgFunc</i>	Return (int) message function
<i>ObjGetClient</i>	Return client of subclass
<i>ObjInit</i>	Initialize object
<i>ObjSetClient</i>	Set client
<i>ObjSendClientGpMsg</i>	Send client a (ptr.) message
<i>ObjSendClientMsg</i>	Send client a (int) message

<i>StrAsMediumInt</i>	String to 16 bit integer
<i>StrExtract</i>	Extract substring from string
<i>StrFromMediumInt</i>	Integer to string
<i>StrHash</i>	Return hash value of string
<i>StrReplaceSubStr</i>	Replace substring in string
<i>StrSqueeze</i>	Removes any character from string
<i>StrToLower</i>	Change case of string to lower
<i>StrToUpper</i>	Change case of string to upper

<i>TskDeInit</i>	Deinitialize task
<i>TskDestroy</i>	Deinitialize task and free space
<i>TskExit</i>	Exit task with code
<i>TskExitWithMsg</i>	Exit task after displaying message
<i>TskInit</i>	Initialize task
<i>TskNew</i>	Initialize task and allocate space

## Thread

A *Thread (Thr)* is used to represent a single thread-of-control (similar to OS/2). However, MS-DOS implements only single threaded programs, therefore there is only one instance of a *Thr*. The only use threads have currently, is as a mechanism for pushing, popping, and invoking exception handlers (in the Ada style). Typically, a program might set up a single exception handler via *Thr* which traps any program logic errors (are triggered with "asserts"). (See also Task page 11 and Exception page 8)

## Tree

A *tree* is a recursive data structure that may contain zero or more children trees and zero or one parent trees.

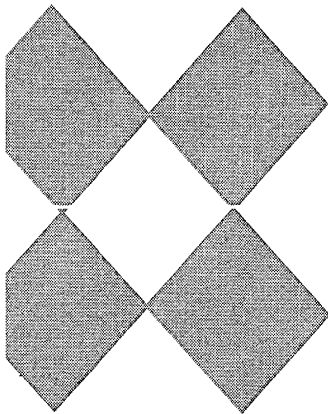
<i>ThrBadParameter</i>	Signal bad function parameter
<i>ThrClear</i>	Clear thread
<i>ThrDisablePushAndPop</i>	Disable further signaling
<i>ThrDiskFull</i>	Signal disk full
<i>ThrDeInit</i>	Deinitialize thread object
<i>ThrEndOfFile</i>	Signal end of file
<i>ThrEnablePushAndPop</i>	Enable signaling
<i>ThrFatalLogicError</i>	Signal program logic error
<i>ThrInit</i>	Initialize thread object
<i>ThrIsFatalError</i>	Is exception non-recoverable?
<i>ThrIsInitialized</i>	Is thread initialized?
<i>ThrOpSysError</i>	Signal system error
<i>ThrOutOfMemory</i>	Signal out of memory
<i>ThrPopCtx</i>	Pop to previous exception handler
<i>ThrPushErrAndReturn</i>	Invoke current exception handler
<i>ThrPushCtx</i>	Push new exception handler
<i>ThrReturnStatus</i>	Signal status condition
<i>ThrWarning</i>	Signal warning

<i>TreAppChild</i>	Append child(ren)
<i>TreAppSibling</i>	Append sibling(s)
<i>TreAsDll</i>	Return tree as linked list
<i>TreAsLel</i>	Return tree as list element
<i>TreAsObj</i>	Return tree as object
<i>TreClient</i>	Return client of tree
<i>TreClientNextSequential</i>	Return next sequential client tree
<i>TreClientDoAllSuccessors</i>	Do function: all successors
<i>TreClientDoBreadthFirst</i>	Do function: breadth first
<i>TreClientDoBranchDepthFirst</i>	Do function: branch depth first
<i>TreClientDoChildren</i>	Do function: children, forwards
<i>TreClientDoChildrenBkws</i>	Do function: children, backwards
<i>TreClientDoDepthFirst</i>	Do function: depth first
<i>TreClientDoDepthFirstBkws</i>	Do function: depth first, backwards
<i>TreClientDoDescBranchDepthFirst</i>	Do function: descendent branches
<i>TreClientDoDescBreadthFirst</i>	Do function: descendent breadth
<i>TreClientDoDescDepthFirst</i>	Do function: descendent depth
<i>TreClientDoDescDepthFirstBkws</i>	Do function: descendent depth
<i>TreClientDoDescLeaves</i>	Do function: descendent leaves
<i>TreClientDoLeaves</i>	Do function: leaves

## Tree

A tree is a recursive data structure that may contain zero or more children trees and zero or one parent trees. (Continued from page 12)

<i>TreClientDoParentsNearestFirst</i>	Do function: nearest parents first
<i>TreClientDoRange</i>	Do function: range
<i>TreClientDoSuccessors</i>	Do function: successors
<i>TreClientFindChild</i>	Do search function: children
<i>TreClientFirstChild</i>	Return client of first child
<i>TreClientLastChild</i>	Return client of last child
<i>TreClientLastLeaf</i>	Return client of last leaf
<i>TreClientNext</i>	Return client of next sibling
<i>TreClientNextUncle</i>	Return client of next uncle
<i>TreClientParent</i>	Return client of parent
<i>TreClientPrev</i>	Return client of previous sibling
<i>TreClientPrevSequential</i>	Return previous client sequentially
<i>TreCut</i>	Cut node(s) from tree
<i>TreCutChildren</i>	Cut children from tree
<i>TreDeInit</i>	Deinitialize tree object
<i>TreDoAllSuccessors</i>	Do function: successors
<i>TreDoBranchDepthFirst</i>	Do function: branches depth first
<i>TreDoBreadthFirst</i>	Do function: breadth first
<i>TreDoChildren</i>	Do function: children
<i>TreDoChildrenBkws</i>	Do function: children backwards
<i>TreDoDepthFirst</i>	Do function: depth first
<i>TreDoDepthFirstBkws</i>	Do function: depth first backwards
<i>TreDoDescBranchDepthFirst</i>	Do function: descendent branches
<i>TreDoDescBreadthFirst</i>	Do function: descendent breadth
<i>TreDoDescDepthFirst</i>	Do function: descendent depth
<i>TreDoDescDepthFirstBkws</i>	Do function: descendent depth
<i>TreDoDescLeaves</i>	Do function: descendent leaves
<i>TreDoLeaves</i>	Do function: leaves
<i>TreDoRange</i>	Do function: range
<i>TreDoSuccessors</i>	Do function: successors
<i>TreFirstChild</i>	Return first child
<i>TreHasChildren</i>	Does tree have any children?
<i>TreHasSiblings</i>	Does tree have any siblings?
<i>TreInit</i>	Initialize tree object
<i>TrelsChild</i>	Is tree a child?
<i>TrelsDirectAncestor</i>	Is related related to another?
<i>TrelsRoot</i>	Is tree the root?
<i>TreInsChild</i>	Insert child(ren)
<i>TreInsSibling</i>	Insert sibling(s)
<i>TreLastChild</i>	Return last child



## Tree

A tree is a recursive data structure that may contain zero or more children trees and zero or one parent trees. (Continued from page 13)

## Vertex

A Vertex (Vtx) is used to represent a node in a directed graph (Grf). A vertex can belong to a single graph. It can access each of its incoming (arrow-end) edges (Edg) and each of its outgoing edges. It can also access all its predecessor vertices and successor vertices. (See also Graph page 9 and Edge page 8)

<i>TreLastLeaf</i>	Return last leaf
<i>TreNew</i>	Allocate and initialize tree object
<i>TreNext</i>	Return next sibling
<i>TreNextSequential</i>	Return next sequential tree
<i>TreNextUncle</i>	Return next uncle
<i>TreParent</i>	Return parent
<i>TrePrev</i>	Return previous sibling
<i>TrePrevSequential</i>	Return previous sequential tree
<i>TreSendMsg</i>	Send a int message to client
<i>TreSendGpMsg</i>	Send a ptr. message to client
<i>VtxAddInEdg</i>	Add incoming edge
<i>VtxAddOutEdg</i>	Add outgoing edge
<i>VtxClear</i>	Clear vertex
<i>VtxConnectToGrf</i>	Connect vertex to graph
<i>VtxCountIn</i>	Count incoming edges
<i>VtxCountOut</i>	Count outgoing edges
<i>VtxDisconnect</i>	Disconnect vertex from graph
<i>VtxDoEdge</i>	Do function: all edges
<i>VtxDoEdgeClients</i>	Do function: clients of all edges
<i>VtxDoInEdge</i>	Do function: incoming edges
<i>VtxDoInEdgeClient</i>	Do function: incoming edges
<i>VtxDoOutEdge</i>	Do function: outgoing edges
<i>VtxDoOutEdgeClient</i>	Do function: outgoing edges
<i>VtxDeInit</i>	Deinitialize vertex object
<i>VtxDestroy</i>	Deinitialize vertex object and free
<i>VtxDisconnectFromGrf</i>	Disconnect vertex from graph
<i>VtxFindOutEdg</i>	Do search function: outgoing edges
<i>VtxFindOutEdgClient</i>	Do search function: outgoing edges
<i>VtxGetClient</i>	Return client of vertex
<i>VtxGetFirstIn</i>	Return first incoming edge
<i>VtxGetFirstOut</i>	Return first outgoing edge
<i>VtxGetGraphLel</i>	Return as list element in graph
<i>VtxGetGrf</i>	Return graph
<i>VtxInGrf</i>	Is vertex in graph?
<i>VtxInit</i>	Initialize vertex object
<i>VtxNew</i>	Initialize vertex, allocate space
<i>VtxRemoveInEdg</i>	Remove incoming edge
<i>VtxRemoveOutEdg</i>	Remove outgoing edge
<i>VtxSendClientMsg</i>	Send message to client

```

struct Node {
    char *name;           /* Node is a specialized kind of Tree */
    Tree tre;             /* Name for each node */
}; typedef struct Node Node;

Node *pNodR = {0};       /* The root node */

Class NodeCls = {0}, *NodTreCls = &NodeCls; /* To inherit from Tree, we need a "class" describing Node */

int main() {
    NodInitializeModule(); /* Initialize classes */
    NodBuildTree();        /* Create a sample set of tree nodes */

    /* "TreClientDo" functions will call a function, NodPrint in these examples,
       and pass the "client" of the tree, a Node pointer in this case, for each tree/node visited */
    /* Print the children nodes of root: a b c */
    TreClientDoChildren( &pNodR->tre, NodPrint ); printf( "\n" ); /* Object-oriented control-structure */
    /* Print the nodes in depth first order: root a.1 a.2 a b c */
    TreClientDoDepthFirst( &pNodR->tre, NodPrint ); printf( "\n" ); /* Object-oriented control-structure */
}

void NodBuildTree( void ) { /* Builds a sample tree of nodes */
    Node *pNod, *pNoda;
    pNodR = NodNew( "root" ); /* Create the root node */
    pNoda = NodNew( "a" );    NodAppChild( pNodR, pNod );
    pNod = NodNew( "b" );    NodAppChild( pNodR, pNod );
    pNod = NodNew( "c" );    NodAppChild( pNodR, pNod );
    pNod = NodNew( "a.1" );  NodAppChild( pNoda, pNod ); /* Note: we are adding to pNoda */
    pNod = NodNew( "a.2" );  NodAppChild( pNoda, pNod ); /* Ditto */
}

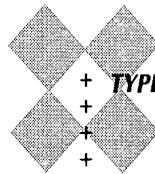
Node *NodNew( char *name ) { /* Allocate memory for a new node and initialize it */
    Node *pNod;
    pNod = (Node *) malloc( sizeof( Node ) );
    TreInit( &pNod->tre, NodTreCls, (char *) pNod ); /* Initialize the tree. TreInit needs a class and instance: the "client" */
    pNod->name = name;
}

void NodAppChild( Node *pNodP, Node *pNodC ) { /* Inherit the Tree function TreAppChild */
    TreAppChild( &pNodP->tre, &pNodC->tre, &pNodC->tre ); /* This adds pNodC as the last child of the parent pNodP */
}

void NodPrint( Node *pNod ) { /* Print a node name given a Node pointer */
    printf( "%s ", pNod->name );
}

void NodInitializeModule( void ) { /* Initialize the class which describes Nodes */
    ClsDefaultInit( NodTreCls ); /* DefaultInit uses a default class description */
}

```



<b>TYPE OF LIBRARY:</b>	Object-Oriented Data Structures, Abstract Data Types, Exception Handler, Date and String
<b>Number of Classes:</b>	18
<b>Number of Functions:</b>	over 300
<b>Compilers:</b>	Microsoft C 5.0+ Quick C 2.0+ Turbo C 2.0
<b>Operating Environments:</b>	DOS Windows OS/2 Xenix Sun Unix
<b>Memory Models:</b>	All models
<b>Version:</b>	2.0



C+Objects is engineered and published by  
Objective Systems  
Architects and Engineers of Software  
2443 Fillmore Street  
San Francisco / California / 94115

C+Objects is a trademark of Objective Systems

**PHONE 415/ 929-0964 FAX 415/ 929-8015**



C C

C C C